Parallel Computation of Ax and $A^T x$

V. Venkatakrishnan[1],

Report RNR-93-001, February 1993

NAS Systems Division
Applied Research Branch
NASA Ames Research Center, Mail Stop T045-1
Moffett Field, CA 94035

February 2, 1993

Abstract

This paper describes how to carry out the matrix vector multiplications Ax and $A^T x$ on parallel computers where A is a sparse matrix arising from the discretization of partial differential equations. Two partitionings of the sparse matrix suitable for parallel computers are discussed. They arise from interpreting the sparse matrix as a graph. One of the techniques partitions the matrix graph by finding edge separators. The other technique partitions the graph by finding vertex separators. We claim that in either case computing $A^T x$ is no more complex than computing Ax. Results from the implementation of the matrix vector multiplications on the Intel iPSC/860 are presented which substantiate the claim. Results from an efficient implementation on the Cray Y/MP-1 are also presented for comparison.

(submitted to The International Journal of High Speed Computing)

[1]Applied Research Branch, Mail Stop T045-1, Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research Center, Moffett Field, CA 94035. The author is an employee of Computer Sciences Corporation. This work was funded under contract NAS 2-12961.

Parallel computation of Ax and A^Tx

V. Venkatakrishnan[1],

Abstract

This paper describes how to carry out the matrix vector multiplications Ax and A^Tx on parallel computers where A is a sparse matrix arising from the discretization of partial differential equations. Two partitionings of the sparse matrix suitable for parallel computers are discussed. They arise from interpreting the sparse matrix as a graph. One of the techniques partitions the matrix graph by finding edge separators. The other technique partitions the graph by finding vertex separators. We claim that in either case computing A^Tx is no more complex than computing Ax. We discuss the data structures necessary to carry out the matrix-vector products for both implementations. Results from the implementation of the matrix vector multiplications on the Intel iPSC/860 are presented which substantiate the claim. Results from an efficient implementation on the Cray Y/MP-1 are also presented for comparison.

1 Introduction

The problem of computing a sparse matrix vector multiplication Ax on parallel computers is an important one. It arises explicitly in many contexts, e.g. when using iterative methods such as conjugate gradient, GMRES [1] etc. in the solution of linear systems. In finite element and finite difference methods, sparse linear systems arise when implicit methods are used. In addition, the computation involved in many explicit methods can be cast as a sparse matrix vector multiplication. This point of view is particularly useful when dealing with finite element or unstructured grids. Some methods for solving linear systems, such as the biconjugate gradient method [2], also require the formation of the transpose matrix vector product A^Tx.

Despite its ubiquity, there is no consensus on how best to carry out the sparse matrix vector multiplications on parallel computers. The purpose of this paper is to show that there are efficient ways to compute Ax and A^Tx. It has been speculated in literature that algorithms which require A^Tx are not suitable for parallel computers. A number of recently developed iterative methods for solving nonsymmetric linear equations, such as the Conjugate Gradients Squared method [3], have been designed explicitly to avoid the A^Tx operation.

A row-oriented representation for the matrix A is chosen for computing Ax and A^Tx. There is a duality between these kernels and the choice of a row-oriented or a column-oriented representation for A. The code for computing A^Tx using the row-oriented data structure is identical to that for computing Ax with the column-oriented data structure and vice versa.

One underlying assumption made in this paper is that the sparse matrix arises from using a bounded stencil in some physical application such as the solution to a partial differential equation. The matrices possess structural symmetry, i.e. if node 1 is connected to node 2, node 2 is connected to node 1 as well. In other words, the sparse matrix graphs that will be considered in this work are generalizations of grid graphs. Whereas in a grid graph each vertex is only connected to its nearest neighbors, here we relax this notion and allow a vertex to be connected in addition to vertices that are distances 2 and higher from the vertex. We shall refer to this broader class of graphs still as generalized grid graphs. Examples of grid graphs are triangular grids, quadrilateral grids and grids composed of a mixture of triangles and quadrilaterals and their counterparts in three dimensions.

[1]Applied Research Branch, Mail Stop T045-1, Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research Center, Moffett Field, CA 94035. The author is an employee of Computer Sciences Corporation. This work was funded under contract NAS 2-12961.

2 Definitions

Grid graphs imply spatial locality. This locality is exploited first by partitioning the vertices. The matrix partitions are derived from these vertex partitions by assigning rows of the matrix to processors. Interprocessor communication is required for accessing the nonlocal entries of the matrix.

A concept that follows from the notion of a grid graph is a *dual graph*. The subgraph consisting of only edges to nearest neighbors can be considered to delineate regions or cells in space. The *dual graph* for a this subgraph is formed by representing each cell as a vertex and each bounding face/edge by an edge.

Given a sparse $m \times n$ matrix, its *adjacency graph* is represented as a set of *Max (m,n)* points (*vertices*) which are connected by a set of lines (*edges*) with orientations. Every off-diagonal entry in the sparse matrix corresponds to a directed edge in the graph where a *directed edge* is an edge with an orientation. The orientation of the edge is from the the vertex corresponding to the row to the vertex corresponding to the column of the off-diagonal entry. If the matrix has a symmetric structure, the orientations of the edges are of no consequence and the resulting graph is an undirected graph. Figure 1 represents an unstructured grid composed of triangles. It can also be interpreted as the adjacency graph corresponding to the matrix with a symmetric structure shown in Figure 2.

A *cell partitioning* is an assignment of cells to processors. This is accomplished by finding vertex separators which form the interpartition boundary. A *vertex separator* is a set of vertices in the adjacency graph whose removal leaves the graph disconnected. In Figure 3, a vertex separator is shown by the darkened line. Figure 3 also shows a local numbering of vertices assigned to each processor. Note that under this partitioning scheme, the vertices on the separator are duplicated. For instance, vertex 4 of processor 0 and vertex 2 of processor 1 represent the same physical vertex.

A *vertex partitioning* is an assignment of vertices to partitions accomplished by computing edge separators. An *edge separator* is the set of edges of the adjacency graph with one endpoint in one vertex partition and the other endpoint in another partition. Figure 4 shows a 2-way vertex partition of the graph of Figure 1 and an edge separator. Vertices 1,2,3 and 4 are assigned to processor 0 and vertices 5, 6 and 7 are assigned to processor 1. In Figure 4, the local numbering of vertices is represented as a tuple, the first entry corresponds to the numbering for processor 0 and the second, for processor 1 for all the vertices incident to the edges of the edge separator. Communication takes place across two rows of vertices on either side of the edge separator. These vertices are formed as the union of vertices adjacent to the edge separator. The lengths of the two lists on two adjacent partitions are not equal in general.

An edge separator can be found for an adjacency graph by a number of techniques described in [4]. The smallest vertex separator can be obtained from the edge separator by several methods also described therein. In the special case of grid graphs, it is possible to compute the vertex separators directly by partitioning the dual graph. Assigning vertices to partitions in the dual graph is equivalent to assigning cells in the original graph.

It is shown in this paper that when the of matrix graph is partitioned either by vertices or by cells, computing $A^T x$ is no more complex than computing Ax. This claim is substantiated with implementation results on the Intel iPSC/860. The trivial way to perform $A^T x$ is to explicitly form the transpose and carry out the matrix-vector multiplication. This is unacceptable for two reasons. The primary reason is that the transpose procedure involves considerable data motion and is expensive. A secondary reason is that the memory requirements are doubled, which cuts down the size of the problem that can be solved. Depending on whether cells or vertices are partitioned, different data structures have to be employed, which are discussed later.

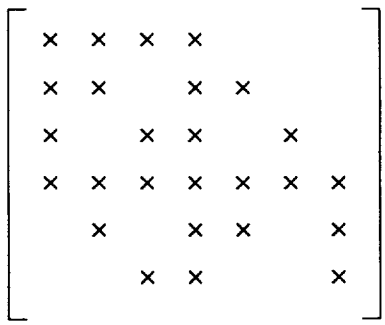

Figure 1: An unstructured grid composed of triangles.

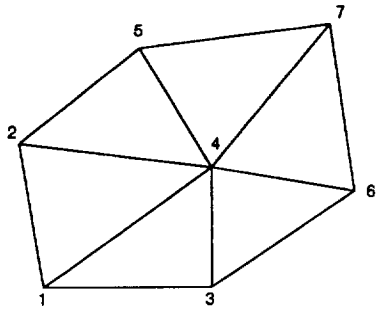

Figure 2: Matrix corresponding to Figure 1.

If there are k degrees of freedom associated with a vertex, the sparse matrix is viewed as a block sparse matrix, where the entries are $k \times k$ blocks. In our application the matrices arise from the solution of two-dimensional Euler equations and therefore, each vertex has four degrees of freedom. Hence, all the operations are carried out on submatrices of size 4×4. On a parallel computer, such as the Intel iPSC/860, one obtains better performance by adopting this viewpoint since there is better cache utilization compared to treating the same matrix entries as scalars.

3 Partitioning of the graph

The mapping of partitions to processors is not a crucial issue as discussed in [5] and therefore a naive mapping of partitions to processors is assumed. Since the partitions are generated recursively the naive assignment consists of mapping partition 0 to processor 0, partition 1 to processor 1 and so on. Therefore, throughout this paper, the words partition and processor will be used interchangeably.

The partitioning step determines the assignment of vertices/cells to processors based on the grid graph. After partitioning, global values of the data structures required to define the grid graph are given local values within each partition. We thus dispense with any references to global indices. Additionally, each local data set contains the information a processor requires for communication at its interpartition boundaries. In our application, the formation of the matrix and the vector is carried out on the parallel computer itself and therefore is not included in the preprocessing. The partitioning and preprocessing are currently done on a workstation.

The recursive spectral partitioning algorithm of Simon [6] is used to partition the grid graphs. Depending on whether cells or vertices are partitioned, different algorithms are obtained for computing the matrix vector product and the transpose matrix vector product. Desirable features of the partitioner are to ensure load balance and to minimize communication costs. The computa-

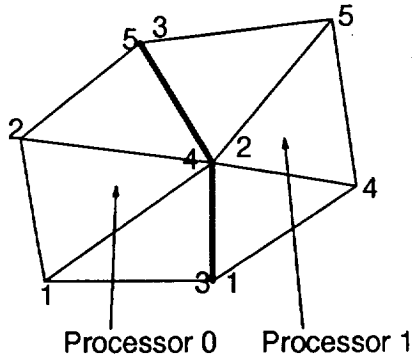

Figure 3: 2-way cell partitioning with a vertex separator.

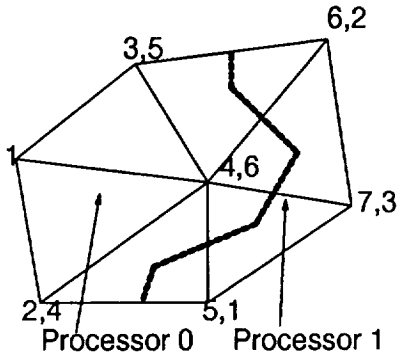

Figure 4: 2-way vertex partitioning with an edge separator.

tional load is proportional to the total number of nonzero entries in the partitioned matrix. In terms of the grid graph, the load is proportional to the sum of the number of vertices and twice the number of edges. However, the recursive spectral partitioning strategy only guarantees nearly equal number of vertices/cells and this only guarantees rough load balance. As mentioned earlier, for vertex partitioning the grid that is partitioned is the grid graph whereas for cell partitioning it is the dual graph.

The spectral algorithm of Pothen et al. [4] derives an edge separator from the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix associated with the graph. The Laplacian matrix consists of the degree of each vertex as the diagonal entry and -1 for each of its neighbors. Pothen et al. [4] have also shown that the separators produced by this algorithm are shorter than those produced by nested dissection. In the present context, separators are of interest since they form the interpartition boundaries. Simon [6] has applied this and other partitioning algorithms to a variety of two-dimensional and three-dimensional problems, arising from finite elements and has shown that the spectral technique yields better partitions in that it produces sub-domains with shorter boundaries. The spectral technique produces uniform, sometimes disconnected sub-domains with short boundaries. Theoretical results by Fiedler (summarized in [4]) show that one of the two sub-domains formed by the spectral partitioning is always connected. Spectral partitioning results in fewer shorter messages and reduced communication cost. This was observed by Venkatakrishnan et al. [5] who found that the spectral partitioning reduced communication costs in comparison with other partitioning strategies in an explicit unstructured Euler solver on the Intel iPSC/860.

4 Data structures

The information required for communication at the interpartition boundaries is precomputed using sparse matrix data structures. The data structures required for cell partitioning are discussed first. The interpartition boundaries are composed of vertex separators. For simplicity, only a nearest-neighbor grid graph is considered. The data structures can be generalized to other grid graphs with larger stencils by employing wider vertex separators. Neighboring subgrids communicate to each other only through their *interpartition boundary vertices* (IBV's) which are shared by the neighboring partitions. The following data structures for each processor facilitate communication across the interpartition boundaries in a general manner:

nadjproc — no. of adjacent processors

iadjproc — list of adjacent processors — length **nadjproc**

ibv — pointers to the cumulative number of IBV's in common with the adjacent processors — length **nadjproc** + 1

nbv — number of boundary vertices in common with processor iadjproc(.). This can be derived from ibv and is not stored; nbv(j) = ibv(j+1) - ibv(j)

nintbv(.,1) — Local indices on current processor — length ibv(nadjproc+1)-1

nintbv(.,2) — Local indices on adjacent processor — length ibv(nadjproc+1)-1

For more details on these data structures, we refer the reader to [5].

With vertex partitioning, the interpartition boundaries consist of edge separators. Here, the IBV's need to be treated differently. The interpartition boundaries now consist of two rows of vertices. Referring to Figure 4, processor 0 receives data from 3 vertices of processor 1 and processor 1 receives data from 3 vertices of processor 0. Therefore, the data structures for each processor consist of:

nadjproc — no. of adjacent processors

iadjproc — list of adjacent processors — length **nadjproc**

ibvs — pointers to the cumulative number of IBV's in common with the adjacent processors for sending messages — length **nadjproc** + 1

ibvr — pointers to the cumulative number of IBV's in common with the adjacent processors for receiving messages — length **nadjproc** + 1

nbvs — number of boundary vertices in common with processor iadjproc(.) for sending messages. This can be derived from ibvs and is not stored; nbvs(j) = ibvs(j+1) - ibvs(j)

nbvr — number of boundary vertices in common with processor iadjproc(i) for receiving messages. This can be derived from ibvr and is not stored; nbvr(j) = ibvr(j+1) - ibvr(j)

nintbvs(.,1) - Local indices on current processor for sending — length ibvs(nadjproc+1)-1

nintbvs(.,2) - Local indices on adjacent processor for receiving — length ibvs(nadjproc+1)-1

nintbvr(.,1) - Local indices on current processor for receiving — length ibvr(nadjproc+1)-1

nintbvr(.,2) - Local indices on adjacent processor for sending — length ibvr(nadjproc+1)-1

5 Uniprocessor implementation of Ax and A^Tx.

Details of the implementation of Ax and A^Tx on a single processors of the Intel iPSC/860 and on the Cray Y/MP-1 are discussed in this section. The implementation on the Cray Y/MP-1 is significantly different from that on a single processor of an Intel iPSC/860. The Cray Y/MP is a vector computer and an efficient implementation should have good vector lengths. The data structure used to store the matrix is edge-based. Associated with each edge are the vertices, n1 and n2 incident to the edge stored in array **nd(.,2)**. The two entries which contain the influence of n2 on n1 and n1 on n2 are stored in array **a (.,2)**. The diagonal entries are stored separately as array **diag(.)**. The matrix vector multiplication is carried out by looping over the edges of the undirected graph for the off-diagonal contributions followed by a loop over the nodes for the diagonal contributions. Vectorization is achieved by coloring the edges of the graph. For simplicity, codes are presented assuming one degree of freedom per vertex. The scalar multiplication should be replaced by a local matrix-vector multiplication when multiple degrees of freedom are associateed with a vertex. In addition, the transpose problem involves a local $k \times k$ matrix transposition before multiplication by the vector, where k is the number of degrees of freedom at each vertex. The pseudocodes for computing Ax is given below:

```
Program mat_mul_cray
Initialize : y(i) = 0,      i = 1, n
do k = 1, kedge
    n1 = nd(k,1)      : vertex 1 incident to the edge k
    n2 = nd(k,2)      : vertex 2 incident to the edge k
    y(n1) = y(n1) + a(k,1)*x(n2)
    y(n2) = y(n2) + a(k,2)*x(n1)
enddo
do j = 1, jnodes
    y(j) = y(j) + diag(j)*x(j)
end do
end
```

The code for computing A^Tx is only slightly different:

```
Program transpose_mat_mul_cray
Initialize : y(i) = 0,      i = 1,n
do k = 1, kedge
    n1 = nd(k,1)      : vertex 1 incident to the edge k
    n2 = nd(k,2)      : vertex 2 incident to the edge k
    y(n2) = y(n2) + a(k,1)*x(n1)
    y(n1) = y(n1) + a(k,2)*x(n2)
enddo
do j = 1, jnodes
    y(j) = y(j) + diag(j)*x(j)
end do
end
```

Table 1 presents the best Cray Y/MP-1 timings for three problem sizes with 4×4 submatrices. The implementations run at about 122-125 megaflops, determined using the Cray hardware performance monitor. The original data structure which is suitable for a cache-based machine such as the

Intel iPSC/860 has the 4×4 submatrix as the first two dimensions and the slowest running dimension over the edges/nodes. For the Cray Y-MP, there was a substantial improvement of about 20% in performance by switching the fastest running dimension to be the over the edges/nodes and the 4×4 submatrices being the last two dimensions in the multi-dimensional arrays. The improvement is mainly due the elimination of bank conflicts which occur with the original data structure.

Table 1: **Performance of Ax and A^Tx on the Cray Y/MP-1.**

Problem size (no. of vertices)	Ax Time (10^{-2} secs.)	A^Tx Time (10^{-2} secs.)
843	0.078	0.080
6019	1.043	1.056
15606	2.803	2.825

The i860, on the other hand, is a cache-based microprocessor so that locality is of paramount importance even on a single processor. The edge-based data structure used on the Cray is not suitable. The usual row-oriented data structure is used to store the matrix in the **ia, ja, a** format, where **ia(.)** contains the cumulative pointers to the start of each row, **ja(.)** contains the column number and **a(.)** contains the entry. The pseudocodes for computing Ax and A^Tx are the same as the ones given in the next two sections except that there is no communication phase.

Table 2: **Performance of Ax and A^Tx on a single processor of the Intel iPSC/960.**

Problem size (no. of vertices)	Ax Time (10^{-2} secs.)	A^Tx Time (10^{-2} secs.)
Row-oriented; last dimension over nodes	1.626	1.493
Row-oriented; first dimension over nodes	1.640	1.490
Edge-based; last dimension over nodes	1.878	1.917

Results are presented in Table 2 from experiments on a small problem with 843 vertices as this is the only problem of the three considered that fits in one processor of the Intel iPSC/860. All the computations are carried out in 64-bit arithmetic. All the codes are compiled with release 2.0 of the Portland Group compiler with "-O4 -Knoieee" options. The best performance is obtained with the row-oriented data structures. The edge-based data structure is about 15% worse for Ax and about 30% worse for A^Tx. Switching the fastest running dimension to be the over the nodes and the 4×4 submatrices being the last two dimensions did not have much impact. This is surprising since locality is lost by employing such a data structure. The best uni-processor implementation of Ax runs at about 5.9 megaflops and A^Tx runs at about 6.5 megaflops in 64-bit arithmetic, all megaflops being Cray Y/MP equivalents. These are obtained by dividing the elapsed times on the Cray and the Intel and multiplying by the Cray megaflop rates. Thus computing A^Tx is faster than computing Ax. This is due to the fact that with A^Tx a scatter operation is employed, whereas with Ax a gather operation is used.

6 Ax and A^Tx with cell partitioning.

This section describes the steps involved in performing the matrix vector multiplications Ax and A^Tx when the cells are assigned to partitions and the interpartition boundaries are composed of vertex separators. The algorithm is presented by considering a simple example. Referring back to the 2-way partition of a simple graph shown in Figure 3, we observe that vertices 3,4 and 5 for processor 0 and vertices 1, 2 and 3 for processor 1 are identical and represent the same physical vertices. Central to our strategy is the *exchange_sum* phase which combines the values of all the vertices over all processors that represent the same physical vertices. For example, referring to Figure 3, if processor 0 has x_4 at vertex 4 and processor 1 has x_2 at vertex 2, at the end of the combine phase, processors 0 and 1 have each the value $x_4 + x_2$ at vertices 4 and 2, respectively. This procedure is easily generalized to handle the case when multiple partitions meet at a vertex.

Under the partitioning scheme, the matrix is itself decomposed into multiple components. Figure 5 illustrates the two components corresponding to Figure 3. Each processor contains five vertices and hence, a 5 x 5 square matrix. The x's refer to entries that are completely local to a processor. The 3 x 3 matrix corresponding to the vertices on the vertex separator is split between the two processors. The symbols other than the x's thus represent partial entries and the full entries are obtained by summing the corresponding partial entries. This splitting is not unique. In our application these entries arise from each processor 'seeing' only a portion of the control volume for the vertices comprising the vertex separator and the splitting is therefore unique.

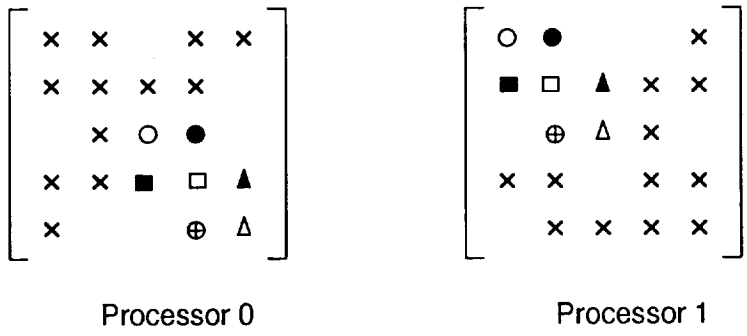

Figure 5: Decomposition of the matrix under the partitioning of Figure 3.

In what follows, it is assumed that the values of x at the corresponding image points are *identical*. This is a reasonable assumption since x is conceivably formed by some procedure that involves communication which ensures this property. For instance, x itself may be the result of a matrix vector multiplication. The computation of Ax in words can be summed up as: For every vertex, sum the non-zero entry multiplied by the value of the vector component corresponding to the entry's column number. The code for matrix vector product Ax is given below:

```
Program mat_mul_cell_part
Initialize : y(i) = 0,      i = 1,n
do j = 1, n
    do l = ia(j),ia(j+1)-1
       y(j) = y(j) + a(l)*x(ja(l))
    enddo
enddo
call exchange_sum (y,n) : add y values at interpartition boundaries
end
```

The first step performs the local computations. At the end of this step, the IBV's hold only partial contributions. The communication phase combines the partial information and yields the full information at the IBV's.

The computation of $A^T x$ in words can be summed up as: For every vertex, compute the product of the entire row by the vector component of the vertex and scatter this vector. Sum all these scattered vectors to get the result. The code for computing $A^T x$ is given below:

```
Program transpose_mat_mul_cell_part
Initialize : y(i) = 0,      i = 1,n
do j = 1, n
    do l = ia(j),ia(j+1)-1
        y(ja(l)) = y(ja(l)) + a(l)*x(j)
    enddo
enddo
call exchange_sum (y,n) : add y values at interpartition boundaries
end
```

Once again the first step performs the local computations. The second step combines the values at the interpartition boundaries. Thus, we observe that the only difference between computing Ax and $A^T x$ is that the former employs a gather operation and the latter, a scatter operation. The communication required is identical.

Results are presented from our implementation of Ax and $A^T x$ for various problem sizes. The matrices in our examples all arise from using triangular grids to solve the Euler equations. Table 3 summarizes our results as the number of processors is varied for three problem sizes. Since the two larger problems do not fit on one processor of the Intel iPSC/860, the scaled speedups are based on the timings obtained with the smallest number of processors required to solve the problems. Elapsed times over ten trials are presented for $Axand A^T x$. Notice in Table 3, that Ax performs slightly worse than $A^T x$. This is true even in the absence of communication on one processor.

Table 3: **Performance of Ax and $A^T x$ on the Intel iPSC/860.**

Problem size (no. of vertices)	No. of procs.	Ax		$A^T x$	
		Time (10^{-2} secs.)	Scaled speedup	Time (10^{-2} secs.)	Scaled speedup
843	1	1.626	1.00	1.493	1.00
	4	0.589	2.76	0.547	2.73
6019	4	5.887	1.00	5.466	1.00
	8	3.275	1.80	3.031	1.80
	16	1.920	3.07	1.799	3.04
	32	1.252	4.70	1.174	4.65
	64	0.870	6.77	0.821	6.66
15606	8	8.128	1.00	7.490	1.00
	16	4.492	1.81	4.168	1.80
	32	2.545	3.19	2.386	3.13
	64	1.701	4.78	1.582	4.73

7 Ax and A^Tx with vertex partitioning.

This section describes the steps involved in performing the matrix vector multiplications Ax and A^Tx when vertices are assigned to processors and the interpartition boundaries are composed of edge separators. The algorithms are presented once again by considering considering the example shown in Figure 4. Under the partitioning scheme, rows of the matrix are assigned uniquely to partitions. Thus the matrix is partitioned into two rectangular matrices of different sizes. Figure 6 illustrates the two components corresponding to Figure 3. Note that in this case, we do not have partial entries since whole rows of the matrix are assigned to one processor or the other.

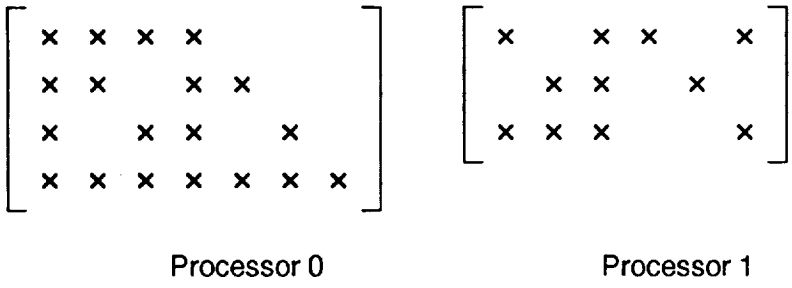

Figure 6: Decomposition of the matrix under the partitioning of Figure 4.

In what follows, it is assumed that the values of x at the corresponding image points are *not identical.* This is again a reasonable assumption since usually the formation of x involves communication (to obtain the values at the image points) followed by some computation. The last step does not ensure that the original point and the image point hold the same value. There needs to be another communication step to ensure this. The code for matrix vector product Ax is given below:

```
Program mat_mul_vertex_part
Initialize : y(i) = 0,      i = 1,n
call exchange (x,n) : exchanges x values at interpartition boundaries
do j = 1, n
    do l = ia(j), ia(j+1)-1
       y(j) = y(j) + a(l)*x(ja(l))
    enddo
enddo
end
```

The first step first communicates data at the interior boundary vertices. For each processor, this step consists of sending data corresponding to a row of interior boundary vertices and receiving data corresponding to the other row of interior boundary vertices. The two messages are not necessarily of the same length. Now each processor has the data necessary for forming Ax for the components that it owns. No more communication is necessary. Comparing with the previous section, we incur nearly the same amount of computation and communication costs.

The code for computing $A^T x$ is given below:

```
Program transpose_mat_mul_vertex_part
Initialize : y(i) = 0, i = 1,n
do j = 1,n
    do l = ia(j),ia(j+1)-1
        y(ja(l)) = y(ja(l)) + a(l)*x(j)
    enddo
enddo
call exchange_sum (y,n) : adds y values at interpartition boundaries
end
```

First, each processor computes a portion of $A^T x$. This step requires no communication since each processor only operates on the rows and vector components assigned to it. However, at the end of this step, the computation of $A^T x$ is not complete. There needs to be a communication step (*exchange_sum* operation) to obtain the correct results at the interpartition boundary vertices. Comparing with the code for computing Ax we observe that $A^T x$ nearly involves the same amount of computation and communication.

Once again results are presented from our implementation of Ax and $A^T x$ for various problem sizes. Table 4 summarizes our results as the number of processors is varied for three problem sizes. Elapsed times over ten trials are shown. Notice that Ax performs slightly worse than $A^T x$.

Table 4: **Performance of Ax and $A^T x$ on the Intel iPSC/860.**

Problem size (no. of vertices)	No. of procs.	Ax		$A^T x$	
		Time (10^{-2} secs.)	Scaled speedup	Time (10^{-2} secs.)	Scaled speedup
843	1	1.626	1.00	1.493	1.00
	4	0.546	2.98	0.512	2.91
6019	4	5.764	1.00	5.254	1.00
	8	3.136	1.84	2.945	1.78
	16	1.814	3.18	1.685	3.12
	32	1.060	5.43	1.004	5.23
	64	0.724	7.96	0.695	7.55
15606	8	7.899	1.00	7.359	1.00
	16	4.274	1.84	3.939	1.86
	32	2.367	3.34	2.188	3.36
	64	1.404	5.63	1.339	5.50

8 Conclusions

In comparing Tables 3 and 4 we draw the following conclusions. Computing A^T is no more expensive than computing Ax; in fact, it is cheaper across the range of problem sizes considered on the Intel iPSC/860. This, we believe, is due to the fact that the transpose problem involves a scatter operation, whereas the matrix vector multiplication involves a gather operation. Recalling the duality discussed earlier, our results on the Intel iPSC/860 indicate that the column-oriented data

structure performs slightly better than the row-oriented data structure when computing the matrix-vector product Ax. Hammond [7] compares two algorithms for computing Ax using row-oriented and column-oriented data structures for A. His implementation results on the CM-2 show that the row-oriented data structure performs significantly better than the column-oriented data structure.

Partitioning the vertices of the original graph as opposed to partitioning cells yields better results. In the case of the matrix vector multiplication, this is because cell partitioning involves floating point operations in the communication phase due to the *exchange_sum* operation whereas vertex partitioning does not. In the case of the transpose problem, both approaches require floating point operations in the communication phase. Still, partitioning of vertices yields slightly better results. Vertex partitioning is more general in that edge separators can always be found, for example, even for complete graphs. Cell partitioning, on the other hand, is less general since it requires vertex separators which, for instance, do not exist for complete graphs.

For general sparse matrices without a symmetric structure but with bounded connectivities, the vertices of the directed graph should be partitioned. In order to ensure good computational load balance, the sum of the total number of directed edges and the number of vertices should be nearly equal across all partitions. This sum equals the total number of nonzero entries in the matrix partitions. This may be difficult to ensure in practice. The interpartition boundaries then consist of edge separators. The algorithms for Ax and A^Tx can then be applied easily. For dense matrices we can still partition the vertices, assign rows of the matrix to processors and apply the algorithms. In this case a complete exchange, wherein each processor broadcasts its component of the vector to all other processors, is required. This can be done following the algorithms outlined by Bokhari [8]. Assigning blocks of the matrix to processors following Fox et al. [9], on the other hand, is more efficient for dense matrices. For random matrices, it pays to follow the strategy for dense matrices if the number of nonzeros is large. If the number of nonzeros is small, partitioning vertices and assigning rows of the matrix to processors is the method of choice.

References

[1] Y. Saad and M.H. Schultz. 1986. GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. stat. Comput.*, 7(3), pp. 856–869.

[2] R. Fletcher. 1976. Conjugate gradient methods for indefinite systems. In G.A. Watson, Editor, *Proc. of the 6-th Biennal Dundee Conf. on Numerical Analysis.* Springer-Verlag.

[3] P. Sonneveld. 1989. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. stat. Comput.*, 10(1), pp. 36–52.

[4] A. Pothen, H. D.Simon, and K.-P. Liou. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11 , pp. 430 – 452.

[5] V. Venkatakrishnan, H.D. Simon, and T.J. Barth. 1992. A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids. *The Journal of Supercomputing*, 6, pp. 117–137.

[6] H. D. Simon. 1991. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, Vol. 2, No. 2/3, pp. 135-148,

[7] S. W. Hammond. 1992. Mapping Unstructured Grid Computations to Massively Parallel Computers. Ph. D. Thesis, Rensselaer Polytechnic Institute, Troy, NY.

[8] S. H. Bokhari.1991. Multiphase complete exchange on a circuit switched hypercube. In *Proc., 1991 International Conf. on Parallel Processing,* vol. 1 (Aug 12-16), CRC Presss, Boca Raton, Fla., pp. I-525–I-529.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker. 1988. *Solving Problems on Concurrent Processors,* Volume 1, Prentice Hall, NJ.